

Complexité des systèmes

Génie informatique et digitalisation

Semestre 2 - Année universitaire 2024-2025

Professeur : Kawtar YOUNSI DAHBI
younsidahbi@emi.ac.ma

Chapitre 1 :Complexité



Complexité algorithmique

- **La complexité algorithmique** désigne une mesure de la quantité de ressources (temps, mémoire, etc.) nécessaires à l'exécution d'un algorithme en fonction de la taille de ses données d'entrée.
- Elle permet d'analyser **l'efficacité d'un algorithme** indépendamment du matériel ou du langage de programmation utilisé.

Pourquoi calculer la complexité ?

- Pour **comparer objectivement** plusieurs algorithmes réalisant la même tâche.
- Pour **prédire le comportement** d'un algorithme sur de grandes données.
- Pour **optimiser les performances** dans les systèmes où les ressources sont limitées.
- Pour **garantir la scalabilité** des systèmes informatiques à long terme.

Les deux types de complexité

- **Complexité en temps** : Mesure le nombre d'opérations élémentaires effectuées par un algorithme en fonction de la taille de l'entrée.
- **Complexité en espace** : Mesure la quantité de mémoire utilisée pendant l'exécution de l'algorithme.



Complexité temporelle

La complexité temporelle d'un algorithme est une information sur son temps d'exécution liée au volume n de données à traiter. La complexité temporelle d'un programme est la complexité temporelle de l'algorithme associé.

La complexité temporelle ne dépend pas :

- des performances du processeur de la machine sur laquelle le programme est exécuté,
- des modes d'accès à la mémoire vive et des modes de stockage,
- du langage de programmation dans lequel l'algorithme est traduit
- du compilateur/interpréteur exécuté.



Complexité temporelle

La **complexité temporelle** d'un algorithme correspond au **nombre total d'opérations élémentaires** qu'il exécute, en fonction de la taille de l'entrée (n).

Les opérations élémentaires

Les **opérations élémentaires** sont les opérations **de base** que l'on considère comme ayant un **coût constant** (elles prennent toujours environ le même temps, peu importe n).

Les principales opérations élémentaires sont :

- Opérations arithmétiques
- Comparaisons
- Affectations simples
- Lecture / écriture

Chacune de ces opérations compte pour 1 (coût = 1)
--

Les séquences

La complexité d'une séquence de n instructions est égale à la **somme des complexités** de chaque instruction qui la compose

$$\text{Coût}_{\text{Séquence}}(d) = \sum_{i=1}^n \text{Coût}_{\text{Instruction}_i}(d)$$

Complexité temporelle

Structure conditionnelle

La complexité d'une instruction conditionnelle est égale à la **somme de la complexité d'évaluation de la condition** et la **plus grande complexité** entre la séquence exécutée si la condition est vraie et celle exécutée si elle est fausse :

$$\text{Coût}_{\text{Choix}}(d) = \text{Coût}_{\text{condition}}(d) + \max(\text{Coût}_{\text{séquence1}}(d), \text{Coût}_{\text{séquence2}}(d))$$

Boucle

La complexité d'une boucle est égale à la **somme** de :

- la complexité de la **séquence d'instructions** qui constitue le **corps** de la boucle, multipliée par le **nombre d'itérations**
- la complexité de l'évaluation de la **condition de sortie**.



Approche asymptotique : Notation O

La complexité asymptotique est le comportement de la complexité d'un algorithme lorsque la taille de son entrée est asymptotiquement grande.

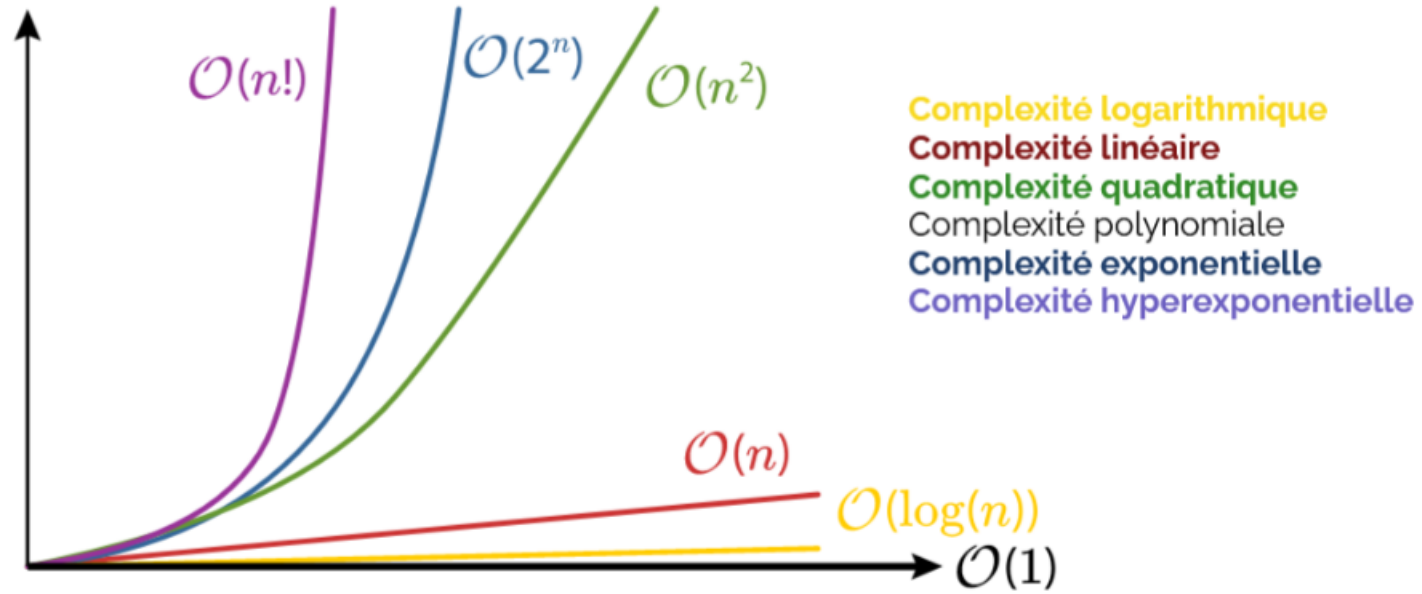
Pour calculer la complexité, on utilise la **notation O** , de la forme $O(f(n))$. Elle permet d'exprimer la complexité d'un algorithme en fonction de la taille de l'entrée, généralement notée n . Le $f(n)$ dans $O(f(n))$ représente ainsi une **fonction mathématique** qui décrit comment la complexité de l'algorithme croît avec n .

L'approche asymptotique permet de :

- **Ignorer les constantes** et les détails d'implémentation pour se concentrer sur l'ordre de grandeur.
- **Abstraire l'analyse** des détails spécifiques à une machine ou une implémentation.
- Obtenir une **évaluation générale, robuste et théorique** des performances.



Type de complexité



Types de complexité

Complexité constante :

On dit qu'un algorithme a une **complexité constante** lorsque le **nombre d'opérations exécutées** est toujours le même, quelle que soit la taille des données en entrée.

On note : $O(1)$

Exemple 1 : Vérifier si un nombre est pair

```
def est_pair(n):  
    return n % 2 == 0
```

Exemple 2: Accéder au dernier élément d'une liste

```
def dernier_element(liste):  
    return liste[-1]
```



Types de complexité

Complexité linéaire:

Un algorithme a une **complexité linéaire** si le nombre d'opérations croît proportionnellement à la **taille de l'entrée**.

- Si la taille de l'entrée double, le temps d'exécution **double** également.
- On note cette complexité : $O(n)$
- Exemple 1 – Afficher chaque élément d'une liste

```
def afficher_liste(liste):  
    for x in liste:  
        print(x)
```

- Exemple 2 – Trouver la valeur maximale d'une liste :

```
def maximum(liste):  
    max_val = liste[0]  
    for x in liste:  
        if x > max_val:  
            max_val = x  
    return max_val
```

Types de complexité

Complexité quadratique:

Un algorithme a une **complexité quadratique** si le **temps d'exécution** est **proportionnel** au **carré** de la **taille de l'entrée**.

- Ce type de complexité apparaît souvent dans les **algorithmes avec deux boucles imbriquées**.
- On note : **$O(n^2)$**

(ou plus généralement **$O(n^k)$** → complexité polynomiale)

Exemple 1 : tri par sélection

```
def tri_selection(liste):  
    for i in range(len(liste)):  
        for j in range(i+1, len(liste)):  
            if liste[j] < liste[i]:  
                liste[i], liste[j] = liste[j], liste[i]
```

Types de complexité

Complexité logarithmique

Un algorithme est dit de **complexité logarithmique** lorsqu'il **réduit la taille du problème en le divisant par deux à chaque étape**.

```
def recherche_binaire(liste, valeur):  
    debut = 0  
    fin = len(liste) - 1  
    while debut <= fin:  
        milieu = (debut + fin) // 2  
        if liste[milieu] == valeur:  
            return milieu  
        elif liste[milieu] < valeur:  
            debut = milieu + 1  
        else:  
            fin = milieu - 1
```



- Pour une liste de **n** éléments, il faut environ **$\log_2(n)$** étapes pour faire la recherche.
- D'où la **complexité en $O(\log n)$** (logarithmique).

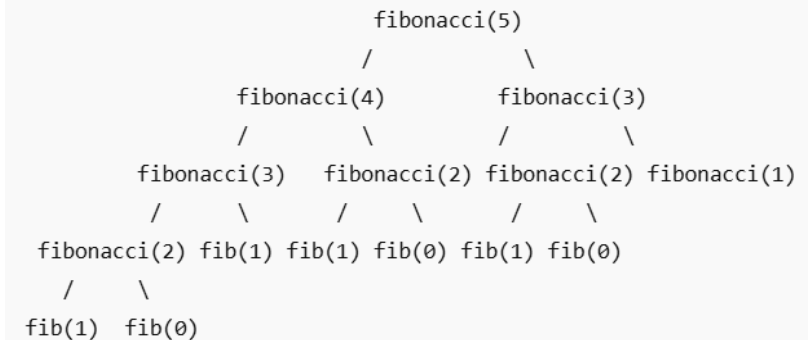
Types de complexité

Complexité exponentielle

Un algorithme a une **complexité exponentielle** si le **nombre d'opérations double à chaque ajout d'un élément dans l'entrée**.

- Très coûteux, **inutilisable pour de grandes tailles d'entrée**.
- Typiquement lié aux **problèmes combinatoires** ou aux **algorithmes récursifs naïfs non optimisés**.

```
def fibonacci(n):  
    if n <= 1:  
        return n  
    return fibonacci(n-1) + fibonacci(n-2)
```



- À chaque appel récursif, la fonction appelle deux sous-problèmes : `fibonacci(n-1)` et `fibonacci(n-2)`.
- Cela crée une arborescence binaire d'appels, avec une croissance exponentielle du nombre d'appels.
- Le nombre total d'appels récursifs est proportionnel à 2^n .

Scénarios d'analyse de la complexité

Quand on analyse un algorithme, on s'intéresse à **sa performance en fonction des données d'entrée**. Mais toutes les entrées ne se comportent pas de la même manière. C'est pourquoi on distingue généralement **trois situations** :

1. Le pire cas – *"le scénario le plus coûteux"*

- C'est la situation où l'algorithme effectue le **plus grand nombre d'opérations**.

Exemple : dans une recherche linéaire, chercher un élément **qui n'existe pas** → il faut parcourir **toute la liste**.

2. Le meilleur cas – *"le scénario le plus favorable"*

- C'est la situation où l'algorithme trouve la solution **très rapidement**, avec le **minimum d'opérations**.

- Cela donne une idée de la **performance optimale**, mais ce n'est **pas toujours représentatif** de la réalité.

- Exemple : dans une recherche linéaire, l'élément recherché est le **premier de la liste** → une seule comparaison suffit → **$O(1)$** .



Scénarios d'analyse de la complexité

3. Le cas moyen – *"le comportement habituel ou typique"*

- C'est la complexité **attendue en général**, si on suppose que les données sont **réparties de façon aléatoire**.
- Il représente la **performance moyenne** de l'algorithme sur des cas réalistes.
- Souvent plus difficile à calculer car il faut **faire des hypothèses** sur les entrées.
- Exemple : en recherche linéaire, en supposant que l'élément est présent **une fois sur deux**, et **uniformément réparti**, on a en moyenne **$n/2$** comparaisons $\rightarrow O(n)$.



Scénarios d'analyse de la complexité

```
def tri_insertion(L):  
    """  
    Trie la liste L en place en utilisant l'algorithme du tri par insertion  
    """  
    for i in range(1, len(L)):  
        cle = L[i]  
        j = i - 1  
        # décaler les éléments plus grands que 'cle' vers la droite  
        while j >= 0 and L[j] > cle:  
            L[j + 1] = L[j]  
            j -= 1  
        L[j + 1] = cle
```

```
L = [3, 1, 4, 2, 5]  
tri_insertion(L)  
# ► Comparaisons et décalages variables selon la position  
# ► Complexité moyenne :  $O(n^2)$ 
```

```
L = [1, 2, 3, 4, 5]  
tri_insertion(L)  
# ► Une seule comparaison par itération, pas de décalage  
# ► Complexité :  $O(n)$ 
```

```
L = [5, 4, 3, 2, 1]  
tri_insertion(L)  
# ► Pour chaque i, on effectue i comparaisons et i décalages  
# ► Nombre total de comparaisons  $\approx n(n-1)/2$   
# ► Complexité :  $O(n^2)$ 
```



Calcul pratique de la complexité

Étape 1 : Identifier les opérations élémentaires

- Ce sont celles qui ont un coût unitaire ($O(1)$)

Étape 2 : Analyser les structures de contrôle

- Repérer les boucles, conditions, fonctions, récursions

Étape 3 : Évaluer le coût de chaque bloc

- Multiplier les coûts des instructions par le nombre de répétitions

Étape 3 : Additionner les coûts

- Somme des coûts des différentes parties de l'algorithme

Étape 5 : Simplifier l'expression finale

- Garder le **terme dominant**
- Exprimer la complexité en **notation O**



Calcul de la complexité : Applications

Exercice 1 – Produit de deux matrices

Écrire un algorithme qui calcule le produit de deux matrices carrées de taille $n \times n$, puis **déterminer sa complexité algorithmique**.

Exercice 2 – Détection de doublons dans une liste

Écrire un algorithme qui retourne les **valeurs en double** dans une liste, puis **déterminer sa complexité algorithmique**.

Exercice 3 – Extraction de l'initiale d'un mot

Écrire un algorithme qui retourne la **première lettre** d'un mot, et **donner sa complexité algorithmique**.



Calcul de la complexité : Applications

Exercice 1 – Produit de deux matrices

```
def produit_matrices(A, B):  
    n = len(A)  
    C = [[0 for _ in range(n)] for _ in range(n)]  
    for i in range(n):  
        for j in range(n):  
            for k in range(n):  
                C[i][j] += A[i][k] * B[k][j]  
    return C
```

Complexité : 3 boucles imbriquées → $O(n^3)$

Exercice 2 – Détection de doublons dans une liste

```
def trouver_doublons(liste):  
    doublons = []  
    for i in range(len(liste)):  
        for j in range(i + 1, len(liste)):  
            if liste[i] == liste[j] and liste[i] not in doublons:  
                doublons.append(liste[i])  
    return doublons
```

Complexité : Deux boucles imbriquées → $O(n^2)$

Exercice 3 – Initiale d'un mot

```
def initiale(mot):  
    return mot[0]
```

Complexité : accès direct à une case mémoire → $O(1)$

Chapitre 2 :Diviser pour régner

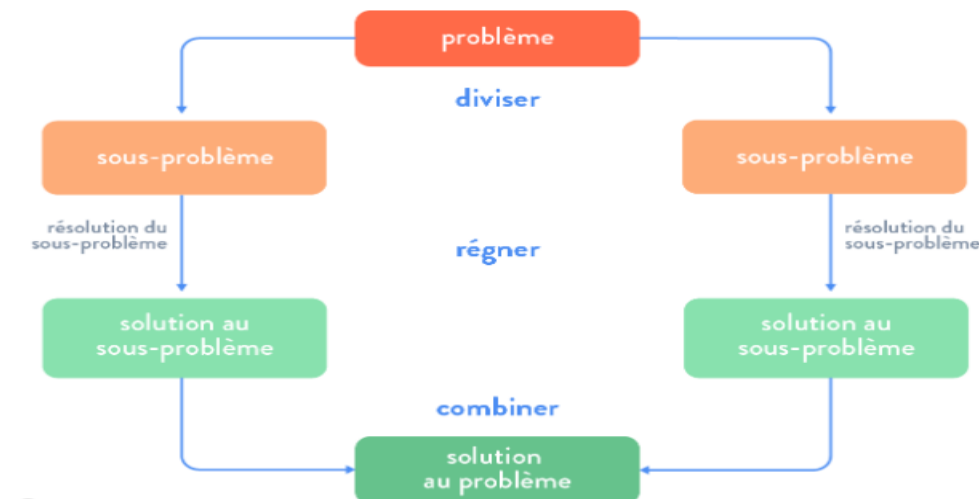


Diviser pour régner

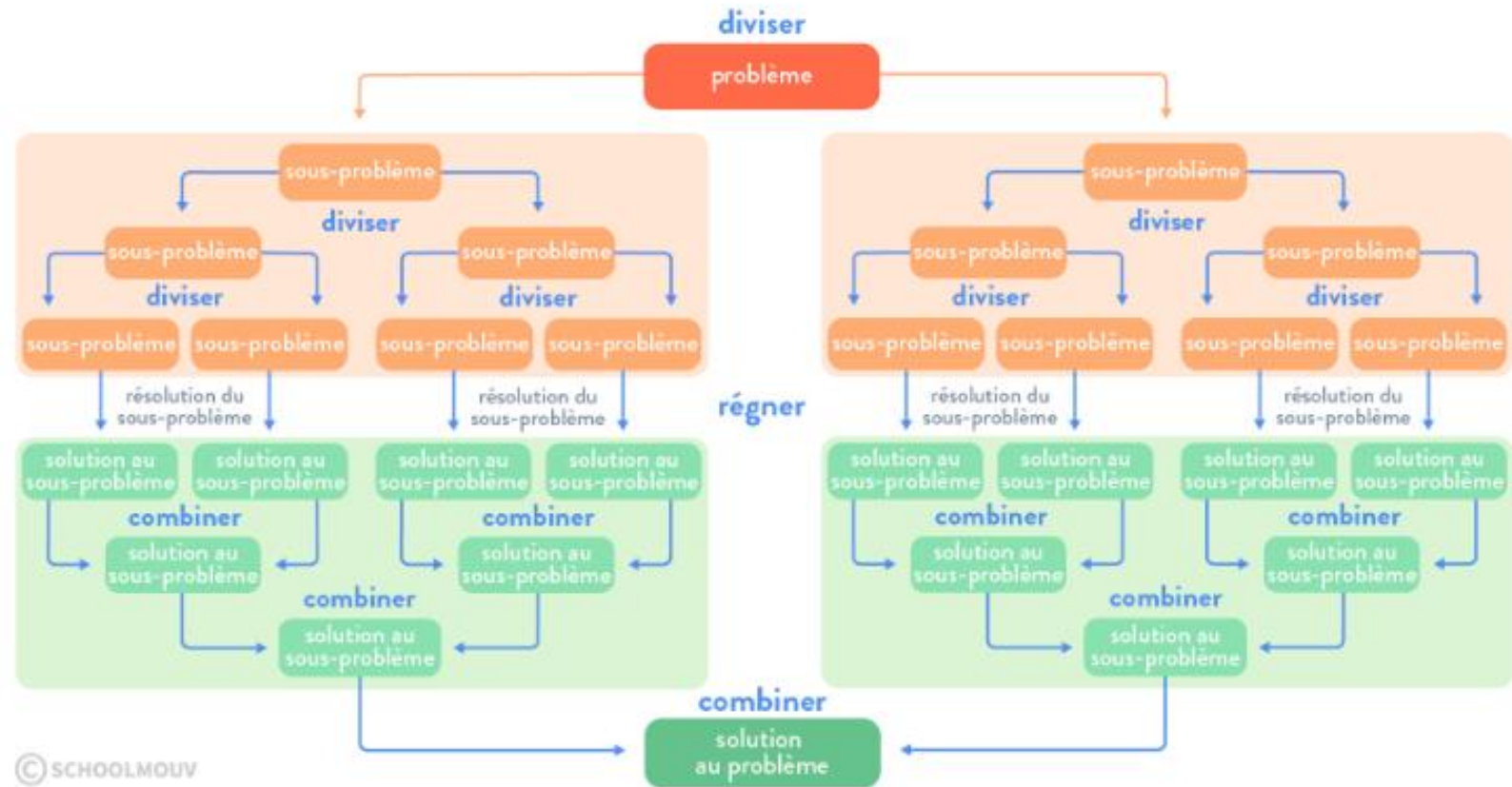
La stratégie **Diviser pour régner** est une méthode classique pour résoudre efficacement des problèmes complexes. Elle consiste à :

1. **Diviser** le problème en sous-problèmes plus petits *et similaires au problème initial*.
2. **Résoudre** récursivement les sous-problèmes.
3. **Combiner** les solutions partielles pour obtenir la solution globale.

Cette approche est particulièrement efficace quand les sous-problèmes sont **de taille réduite** et **indépendants** les uns des autres. Elle permet souvent d'améliorer la complexité d'un algorithme.



Diviser pour régner

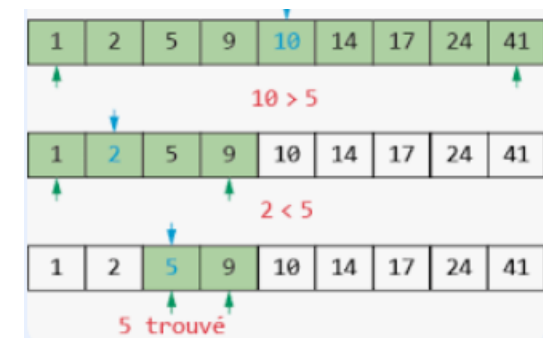


Recherche dichotomique

La recherche dichotomique (ou binaire) est un algorithme de recherche rapide utilisé sur des listes triées. Elle est basée sur la stratégie **Diviser pour régner**.

Principe :

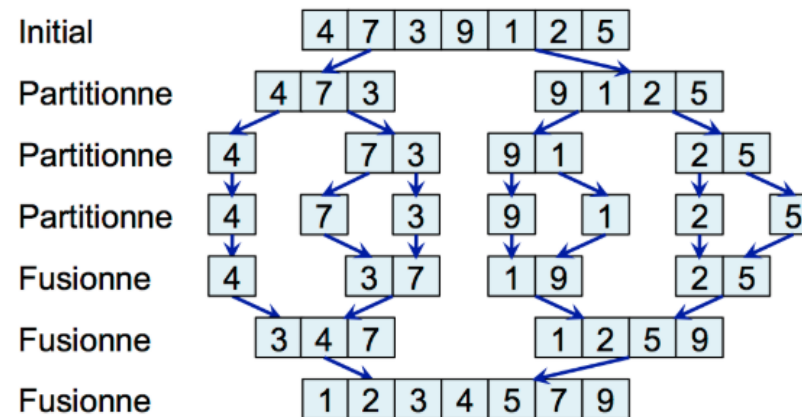
- Diviser la liste en deux.
- Comparer l'élément du milieu avec la valeur recherchée :
 - Si c'est égal : on a trouvé.
 - Si la valeur recherchée est inférieure : on cherche dans la moitié gauche.
 - Si elle est supérieure : on cherche dans la moitié droite.
- Répéter le processus sur la moitié restante.



Tri par fusion

Le **tri par fusion** est un algorithme de tri efficace basé sur la stratégie **Diviser pour régner**.
Principe :

- Diviser la liste en deux moitiés.
- Trier récursivement chaque moitié.
- Fusionner les deux moitiés triées en une seule liste triée



Chapitre 3 : Algorithmes gloutons



Problème d'optimisation combinatoire

Un **problème d'optimisation combinatoire** peut être formulé de la manière suivante :

- **Entrée** : un ensemble fini de configurations possibles
- **Fonction objectif** : une fonction qui associe un coût ou une valeur à chaque configuration.
- **Contraintes** : un ensemble de règles que toute solution admissible doit respecter.
- **Objectif** : identifier une configuration valide qui minimise ou maximise la fonction objectif.

Caractéristiques d'un problème d'optimisation

- Une **solution** est généralement un sous-ensemble des données du problème.
- Il existe souvent **plusieurs solutions admissibles** (c'est-à-dire conformes aux contraintes).
- À chaque solution admissible est associée une **valeur**, représentant un **coût** (à minimiser) ou un **gain** (à maximiser).

Ainsi, un problème d'optimisation ne consiste pas seulement à trouver une solution valide, mais à identifier **la meilleure solution possible**, selon le critère défini par la fonction objectif.



Stratégie Brute force

La stratégie **brute force** (ou **force brute**) consiste à **explorer toutes les solutions possibles** à un problème pour **trouver la meilleure**.

C'est une approche **exhaustive**, simple à comprendre, mais souvent très **coûteuse en temps de calcul**.

Principe :

- **Générer toutes les combinaisons** ou tous les cas possibles.
- **Tester chaque solution**, une par une.
- **Conserver** la meilleure solution rencontrée.

Avantages :

- **Très simple à implémenter**
- **Toujours correct** (trouve la solution optimale)

Inconvénients :

- **Très lent** pour les problèmes avec beaucoup de possibilités (explosion combinatoire)
- **Peu pratique** dès que **n devient grand**



Algorithme glouton

La méthode gloutonne est un paradigme algorithmique qui consiste à construire une solution **étape par étape**, en choisissant **le meilleur choix local** à chaque étape, **sans revenir en arrière**.

Principe :

- Partir d'une solution vide.
- À chaque étape, faire le choix optimal immédiat (le plus avantageux à court terme) sans jamais revenir en arrière
- Répéter jusqu'à ce que la solution soit complète.



Algorithme glouton

Exemple :

On dispose d'une **salle** pouvant être louée pour des durées variables.

Un ensemble de **n demandes de location** est reçu.

Chaque demande i est définie par :

- une **date de début**
- une **date de fin**

•**Objectif :**

Sélectionner un **sous-ensemble de demandes à accepter** tel que :

- aucun chevauchement** ne se produise
- et le **nombre de demandes acceptées soit maximal**.

Ce problème est un **problème classique d'ordonnancement d'intervalles**



Algorithme glouton

Activité	Heure de début (d_i)	Heure de fin (f_i)
A1	08h00	10h00
A2	09h00	11h00
A3	08h00	12h00
A4	11h00	13h00
A5	13h00	14h00
A6	11h00	14h00
A7	12h00	15h00
A8	13h00	16h00
A9	09h00	17h00

Stratégie gloutonne :

1. Trier les demandes par **ordre croissant de date de fin**.
2. Initialiser la sélection avec la première demande du tri.
3. À chaque étape, **ajouter la prochaine demande** si elle **ne chevauche pas** la dernière sélectionnée.

• Résultats

- **Activités sélectionnées** : A1, A4, A5
- **Nombre total d'activités compatibles** : 3

Algorithme glouton

Quand utiliser la méthode gloutonne ?

La méthode gloutonne fonctionne si le problème satisfait deux propriétés fondamentales :

1. Sous-structure optimale

Une solution optimale globale contient des solutions optimales aux sous-problèmes.

2. Propriété du choix glouton

Le meilleur choix local mène à la solution optimale globale, sans devoir revenir sur les décisions précédentes.



Algorithmes gloutons

Exemple 1 – Rendu de monnaie

Contexte :

On dispose de pièces de valeurs $\{1, 5, 10, 25, 50\}$.

On souhaite rendre une somme donnée avec le **minimum de pièces**.

Objectif :

Minimiser le nombre total de pièces utilisées.

Choix glouton :

À chaque étape, choisir la **plus grande pièce** possible sans dépasser la somme restante.

Le choix donne une solution optimale **si les pièces sont bien conçues** (comme les euros ou dollars).

Mais ce n'est **pas toujours optimal** si on utilise d'autres pièces (ex. : $\{1, 3, 4\}$ pour rendre 6 $\rightarrow$ le glouton donne $4 + 1 + 1 = 3$ pièces, mais la meilleure solution est $3 + 3 = 2$ pièces).



Algorithmes gloutons

Exemple 2 – Sac à dos fractionnaire

Contexte :

On a des objets avec poids et valeurs.

On peut **fractionner les objets** pour remplir un sac de capacité W .

Objectif :

Maximiser la **valeur totale** transportée dans le sac.

Choix glouton :

Choisir les objets par **valeur/poids décroissante**, et **remplir progressivement le sac**, en fractionnant si nécessaire.



Algorithmes gloutons

Exemple 2 – Sac à dos fractionnaire

Principe général :

- Calculer la valeur par unité de poids pour chaque objet.
- Trier les objets selon ce ratio en ordre décroissant.
- Initialiser la capacité restante du sac.
- Parcourir les objets triés :
 - Si l'objet entier peut tenir, le prendre complètement.
 - Sinon, prendre la fraction maximale possible de l'objet pour remplir le sac.
- La valeur totale accumulée est la solution optimale.



Algorithmes gloutons

Exemple 4 – Algorithme de Dijkstra

Contexte :

On a un graphe pondéré (avec poids positifs sur les arêtes).

On veut trouver le **plus court chemin** entre un sommet de départ et tous les autres sommets.

Objectif :

Minimiser la **distance totale** entre le sommet de départ et les autres sommets.

Choix glouton :

Toujours choisir le **sommet non visité avec la plus petite distance connue**, puis mettre à jour les distances de ses voisins.

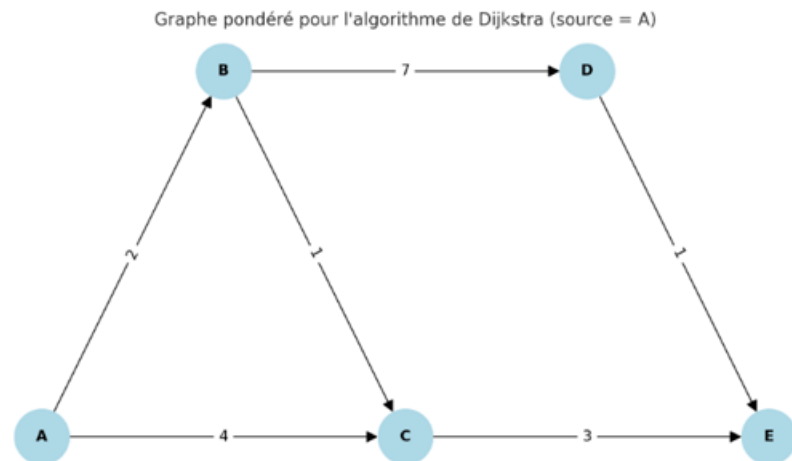


Algorithme gloutons

Exemple 4 – Algorithme de Dijkstra

Principe général

1. Initialiser toutes les distances à l'infini, sauf la source (distance = 0).
2. Créer un ensemble des sommets non encore traités.
3. Tant que cet ensemble n'est pas vide :
 1. Sélectionner le sommet u ayant la **plus petite distance temporaire**.
 2. Marquer u comme traité.
 3. Pour chaque voisin v de u non encore traité :
 1. Si le chemin passant par u vers v est plus court, **mettre à jour la distance** de v .
4. Les distances finales représentent le coût minimum pour aller de la source à chaque sommet.



Algorithme gloutons

Exemple 5 – Arbre couvrant minimal (Kruskal)

Contexte :

On a un graphe pondéré.

On veut relier tous les sommets avec un sous-ensemble d'arêtes **sans cycle**, et **avec poids total minimal**.

Objectif :

Construire un **arbre couvrant de poids minimal**.

Choix glouton :

Ajouter les arêtes **par poids croissant**, en évitant les cycles.



Algorithme gloutons

Exemple 5 – Arbre couvrant minimal (Kruskal)

Principe général :

- Initialiser un ensemble vide pour stocker les arêtes de l'arbre couvrant.
- Trier toutes les arêtes du graphe par poids croissant.
- Parcourir les arêtes triées :
 - Si l'ajout de l'arête ne forme pas de cycle (via Union-Find), l'ajouter à l'arbre couvrant.
- Répéter jusqu'à avoir exactement $n-1$ arêtes (où n est le nombre de sommets).
- L'arbre final est l'arbre couvrant de poids minimal.



Algorithme glouton

Exemple 6 – Arbre couvrant minimal (Prim)

Contexte

On travaille sur un graphe non orienté et pondéré.

L'objectif est de relier tous les sommets avec un sous-ensemble d'arêtes sans former de cycle, et avec un poids total minimal.

Objectif

Construire un **arbre couvrant de poids minimal** (Minimum Spanning Tree) en partant d'un sommet initial.

Choix glouton

À chaque étape, **ajouter l'arête de poids minimal** qui relie un sommet déjà visité à un sommet non encore visité.

On construit l'arbre **progressivement** à partir d'un sommet donné.



Algorithme gloutons

Exemple 6 – Arbre couvrant minimal (Prim)

Principe général

- Démarrer à partir d'un **sommet arbitraire**.
- Utiliser une **file de priorité** (souvent un min-heap) pour sélectionner les **arêtes de poids minimal** connectant l'arbre courant à un nouveau sommet.
- À chaque étape :
 - Sélectionner l'arête de plus **petit poids** qui connecte un sommet **déjà visité** à un sommet **non visité**.
 - Marquer le nouveau sommet comme **inclus** dans l'arbre couvrant.
 - Répéter jusqu'à ce que tous les sommets soient inclus.



Chapitre 4 : Programmation dynamique



Programmation dynamique

La programmation dynamique est une méthode utilisée pour résoudre des problèmes complexes en les décomposant en **sous-problèmes plus simples**.

Elle est particulièrement efficace lorsque **les mêmes sous-problèmes se répètent** plusieurs fois.

Principe :

- Résoudre **une seule fois** chaque sous-problème
- **Stocker le résultat** pour le réutiliser (au lieu de recalculer)
- Construire la solution finale à partir des résultats **déjà connus**

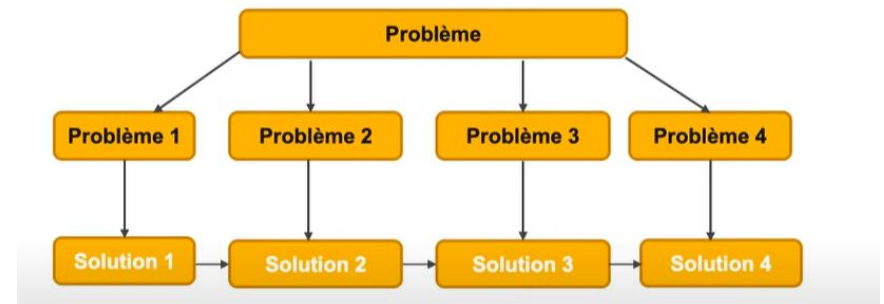
Conditions pour utiliser la programmation dynamique :

1. Chevauchement des sous-problèmes :

Certains sous-problèmes apparaissent plusieurs fois dans l'arbre de récursion.

2. Sous-structure optimale :

La solution globale peut être construite à partir de solutions optimales des sous-problèmes.



Programmation dynamique

Deux approches :

- **Mémoïsation (top-down) :**

On part du problème principal, on le divise récursivement, et on **mémore les résultats**.

- **Tabulation (bottom-up) :**

On commence par les **cas de base**, et on **remonte progressivement** jusqu'à la solution finale.

Pourquoi c'est efficace ?

- Évite les **calculs répétitifs**
- Permet de résoudre certains problèmes en **temps polynomial**
- Utilisable quand les algorithmes naïfs sont **trop lents (exponentiels)**



Programmation dynamique

Exemple 1 – Suite de Fibonacci

Problème :

Calculer le n-ième terme de la suite de Fibonacci définie par :

$$F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)$$

Pourquoi programmation dynamique ?

- Les calculs se répètent dans la version récursive naïve (ex : $F(2)$, $F(3)$...).
- La programmation dynamique permet de mémoriser les résultats intermédiaires.
- Évite les appels redondants et diminue la complexité de $O(2^n)$ à $O(n)$.

Principe général :

- On initialise un tableau dp de taille $n+1$.
- On remplit les valeurs de bas en haut (approche bottom-up).
- À chaque étape : $dp[i] = dp[i-1] + dp[i-2]$



Programmation dynamique

Exemple 2 – Problème du sac à dos 0/1 (Knapsack)

Problème :

Étant donné un ensemble d'objets avec un poids et une valeur, et une capacité maximale, maximiser la valeur transportée sans dépasser la capacité (on ne peut pas fractionner les objets).

Principe général :

- Utiliser un tableau $dp[i][w]$ représentant la valeur maximale obtenue avec les i premiers objets et une capacité w .
- Choix entre : prendre ou ne pas prendre l'objet i .

Étapes :

1. Initialisation :

1. $dp[0][w] = 0$ pour tout w

2. Pour chaque objet i de 1 à n :

1. Pour chaque capacité w de 0 à W :

1. Si $poids[i-1] > w \rightarrow dp[i][w] = dp[i-1][w]$
2. Sinon $\rightarrow dp[i][w] = \max(dp[i-1][w], dp[i-1][w - poids[i-1]] + valeur[i-1])$

3. Le résultat est $dp[n][W]$

Programmation dynamique

Exemple 3 – Algorithme de Bellman-Ford

Problème :

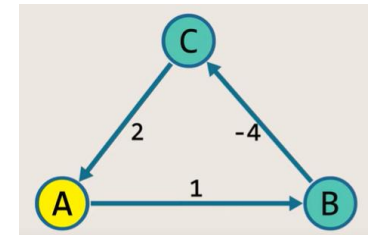
- Trouver les plus courts chemins depuis un sommet source dans un graphe orienté pondéré.
- Le graphe peut contenir des poids négatifs.

Pourquoi programmation dynamique ?

- Les sous-problèmes consistent à calculer les distances minimales avec au plus k arêtes.
- À chaque itération, on utilise les résultats précédents (relation de dépendance entre étapes).
- On évite les recalculs en mettant à jour directement les distances dans un tableau.

Principe général :

- Initialiser toutes les distances à ∞ sauf celle de la source (0).
- Répéter $n - 1$ fois (n = nombre de sommets) :
 - Pour chaque arête (u, v, poids) :
 - Si $\text{dist}[u] + \text{poids} < \text{dist}[v] \rightarrow$ mettre à jour $\text{dist}[v]$.
- Vérifier la présence d'un cycle négatif en refaisant une dernière passe :
 - Si une mise à jour est encore possible, il existe un cycle de poids négatif atteignable.



Programmation dynamique

Exemple 4 – Longest Common Subsequence (LCS)

Problème :

Trouver la plus longue sous-séquence commune entre deux chaînes de caractères.

Exemple :

X = "ABCBDA B"

Y = "BDCAB"

LCS = "BDAB"

Pourquoi programmation dynamique ?

- Plusieurs sous-problèmes se recoupent (ex : comparer préfixes).
- La solution optimale dépend de solutions optimales de sous-chaînes plus courtes.

Principe général :

- On construit une matrice $dp[i][j]$ où chaque case représente la longueur du LCS entre $X[0..i]$ et $Y[0..j]$.
- Si les caractères correspondent, on ajoute 1 à la solution précédente en diagonale.

Étapes :

1. Initialisation :

- $dp[0][*] = dp[*][0] = 0$

2. Pour i de 1 à m , j de 1 à n :

- Si $X[i-1] == Y[j-1] \rightarrow dp[i][j] = dp[i-1][j-1] + 1$
- Sinon $\rightarrow dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$

3. Le résultat est $dp[m][n]$